behaviac 深入解析

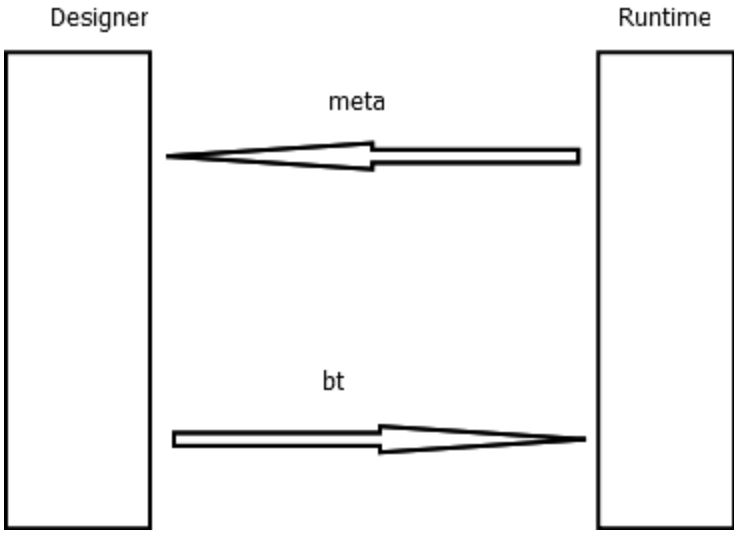
腾讯游戏
Tencent Games

https://github.com/TencentOpen/behaviac

目录

1 概述

behaviac 是我们对行为树（Behavior Tree）的一个实现方案。该方案包括一个编辑器（Designer）和一个 Runtime 的库（c++/C#）。编辑器（designer）用来编辑和调试 BT，Runtime 的库用来解释和执行编辑器（Designer）生成的 BT。



本文重点剖析 behaviac 内部实现的技术细节。着重点在于 C++ Runtime 的实现，C# Runtime 的实现大同小已。

2 元信息

游戏中负责运行行为的实体称谓 Agent，每一个 Agent 都存有一定的数据或者能执行一定的动作。这样的数据被称谓属性（property），这样的动作被称谓方法（method），所谓的元信息（meta）就是对一个 Agent 的描述，包括其属性（property）和方法（method）等。Runtime 的库（C++ lib）或 C#的库产生一个描述 agent 元信息（meta）的 xml 文件。

Behaviac 的实现基于反射系统。Behaviac 使用了一个基于宏的反射系统来注册相关数据类型及属性和方法，并且把这些信息（所谓的元信息）导出到一个 xml 文件从而 Designer 可以使用。而在执行 BT 的过程中，通过该反射系统来访问相应的属性和方法。下图是该反射系统注册的宏：

```
BEGIN_PROPERTIES_DESCRIPTION(BtSnowNinja)
    REGISTER_PROPERTY(targetBrain).SETNETROLE(behaviac::NET_ROLE_NONAUTHORITY);

    REGISTER_METHOD(ClearFireJumpFlag);
    REGISTER_METHOD(FindTarget).ADDPARAM("type").ADDPARAM("alertRange");
    REGISTER_METHOD(DistanceTo).ADDPARAM("target");
    REGISTER_METHOD(Fire).ADDPARAM("target").ADDPARAM("fireRange").ADDPARAM("fire_scale");
    REGISTER_METHOD(Aim).ADDPARAM("target");
    REGISTER_METHOD(Move).ADDPARAM("target").ADDPARAM("farRange").ADDPARAM("closeRange");
    REGISTER_METHOD(RandomJump).ADDPARAM("jumpRange");
    REGISTER_METHOD(Wander);

    REGISTER_METHOD(Health);
    REGISTER_METHOD(SetDebugHealth).SETNETROLE(behaviac::NET_ROLE_NONAUTHORITY);

    REGISTER_METHOD(GetCollidedObject);
    REGISTER_METHOD(Heal);
    REGISTER_METHOD(TakeDamage);

    REGISTER_EVENT("Collided");

END_PROPERTIES_DESCRIPTION()
```

其中 `REGISTER_PROPERTY`，`REGISTER_METHOD`，`REGISTER_EVENT` 分别用来注册属性，方法和事件。特别的 class 的 static 的属性和方法也是支持的。这些宏的实现基于 template 以及 template specialization 从而可以从属性或方法的原型而推导出具体的类型信息来。
由于 C#对反射系统原生的支持，C#的实现则更加直接，用下述的 Attribute 来修饰相应的类型，属性或方法：

- TypeMetaInfoAttribute
- MemberMetaInfoAttribute
- MethodMetaInfoAttribute
- EventMetaInfoAttribute
- ParamMetaInfoAttribute

需要指出和强调的是，元信息 meta **由 Runtime 生成**，Designer 只是读入并且利用该元信息来‘描述’BT，元信息不是由 Desinger 生成的。当然某些情况下也可以手工的修改该元信息来包括一些 Runtime 还没有实现的一些属性或方法从而做一些实验。

元信息是仅供 Designer 使用的，Runtime 不需要使用元信息，元信息只需要**开发版本**中当 Agent 有改变的时候导出一次就可以，不需要每次都导出。最终的发布版本中不需要导出元信息 meta 文件。然而在开发版本中，既可以在一个特殊的工具中导出元信息，也可以每次运行开发版游戏的时候导出元信息，这样子可以确保任何 Agent 的改动都可以更新元信息 meta 文件，只要 Agent 没有改变，导出的元信息 meta 文件也将是一样的。

另外，如果是通过运行开发版游戏产生或者更新导出元信息，当修改了 Agent 后，原本根据旧的元信息产生的 BT 就不能运行了，有人会抱怨由于 BT 不能运行导致开发版的游戏 crash 而不能产生元信息。解决方法在于**不用纠结于游戏的 crash**，只要把产生元信息的函数调用放在使用 BT 前就可以了，因为产生元信息只依赖 Agent 的注册，不依赖于 BT。当然也可以考虑添加某个参数专门用来产生或更新元信息。

2.1 生成元信息的 xml 文件

生成元信息的 xml 文件则如图：

```
<metas>
    <agents>
        <agent classfullname="behaviac::Agent" inherited="true" DisplayName="" Desc="" />
        <agent classfullname="behaviac::World" inherited="true" DisplayName="" Desc="" />
        <agent classfullname="test_ns2::PlayerTestBase" inherited="true" DisplayName="" Desc="">
            <Member Name="Property1" DisplayName="" Desc="" Type="signed int" Class="test_ns2::PlayerTestBase" />
            <Member Name="Property2" DisplayName="" Desc="" Type="bool" Class="test_ns2::PlayerTestBase" />
            <Member Name="Property3" DisplayName="" Desc="" Type="float" Class="test_ns2::PlayerTestBase" />
            <Member Name="static_Property4" DisplayName="" Desc="" Type="bool" Class="test_ns2::PlayerTestBase" Static="true" />
            <Method Name="action0" DisplayName="" Desc="" Class="test_ns2::PlayerTestBase" ReturnType="void" />
            <Method Name="action1" DisplayName="" Desc="" Class="test_ns2::PlayerTestBase" ReturnType="void">
                <Param DisplayName="" Desc="" Type="signed int" />
            </Method>
            <Method Name="static_action2" DisplayName="" Desc="" Class="test_ns2::PlayerTestBase" Static="true" ReturnType="signed int" />
            <Method Name="static_action3" DisplayName="" Desc="" Class="test_ns2::PlayerTestBase" Static="true" ReturnType="void">
                <Param DisplayName="" Desc="" Type="signed int" />
            </Method>
            <Method Name="method_dawn" DisplayName="" Desc="" Class="test_ns2::PlayerTestBase" ReturnType="bool">
                <Param DisplayName="" Desc="" Type="signed int" />
            </Method>
            <Method Name="event_explode" DisplayName="" Desc="" Flag="namedevent" Class="test_ns2::PlayerTestBase" ReturnType="bool" />
            <Method Name="event_end" DisplayName="" Desc="" Flag="namedevent" Class="test_ns2::PlayerTestBase" Static="true" ReturnType="bool" />
        </agent>
        <agent classfullname="test_ns::AgentBase" inherited="true" DisplayName="Agent基类，用来显示用的" Desc="Agent基类的说明">
            <Member Name="Base_Property1" DisplayName="基类中的Property1" Desc="基类中的Property1的说明" Type="signed int" Class="test_ns::AgentBase" />
            <Member Name="Base_Property2" DisplayName="基类中的Property2" Desc="基类中的Property2的说明" Type="bool" Class="test_ns::AgentBase" />
```

该 xml 文件包含了：

- 每个 Agent 的类型信息
- Member，成员属性
- Method，方法或事件

而每个类型，成员属性和方法都包含了名字（Name），显示名字（DisplayName），描述（Desc）以及其他一些信息。特别的每个 Agent 类型的名字是包含了 namespace 的全名。而 DisplayName 和 Desc 是供编辑器显示用的，这样，更有意义的中文名字，更多的说明信息等都可以在编辑器中显示。

2.2 编辑器中对元信息的使用

编辑器打开一个 workspace 后首先根据配置的元信息的 xml 文件在 "C:\Users\jonli\AppData\Local\Temp\Behaviac\"生成一个 c#的源文件，该 c#的源文件是一些从 Behaviac.Design.Agent 继承的类的集合，每个这样的类实际上是 xml 中相应的类型的一个 c#的描述。随后，编辑器使用该 c#文件生成 "C:\Users\jonli\AppData\Local\Temp\Behaviac\XMLPluginBehaviac.dll"并且加载该 dll 从而通过 c#的反射系统而方便的获取并且使用所有的类型信息。

```
namespace XMLPluginBehaviac
{
    [Behaviac.Design.ClassDesc("behaviac::World", "Agent", true, "behaviac::World", "behaviac::World")]
    public class behaviac_World : Behaviac.Design.Agent
    {

    }

    [Behaviac.Design.ClassDesc("framework::GameObject", "Agent", true, "framework::GameObject", "framework::GameObject")]
    public class framework_GameObject : Behaviac.Design.Agent
    {
        [Behaviac.Design.MemberDesc("framework::GameObject", false, false, "unsigned int", "HP", "HP")]
        public uint HP;

        [Behaviac.Design.MemberDesc("framework::GameObject", false, false, "signed long", "age", "age")]
        public long age;


        [Behaviac.Design.MethodDesc("framework::GameObject", false, false, false, false, "void", "GoStraight", "GoStraight")]
        public delegate void GoStraight(
            [Behaviac.Design.ParamDesc("signed int", "speed", "speed", "")]
            int param0
            );

        [Behaviac.Design.MethodDesc("framework::GameObject", false, false, false, false, "signed int", "TurnTowardsTarget", "TurnTowardsTarget")]
        public delegate int TurnTowardsTarget(
            [Behaviac.Design.ParamDesc("float", "turnSpeed", "turnSpeed", "")]
            float param0
            );

        [Behaviac.Design.MethodDesc("framework::GameObject", false, false, false, false, "bool", "alignedWithPlayer", "alignedWithPlayer")]
        public delegate bool alignedWithPlayer(

            );

        [Behaviac.Design.MethodDesc("framework::GameObject", false, false, false, false, "bool", "playerIsAligned", "playerIsAligned")]
        public delegate bool playerIsAligned(

            );

        [Behaviac.Design.MethodDesc("framework::GameObject", false, false, false, false, "bool", "projectileNearby", "projectileNearby")]
        public delegate bool projectileNearby(
            [Behaviac.Design.ParamDesc("float", "radius", "radius", "")]
            float param0
            );
```

2.3 关于 method 的一些说明

例如 REGISTER_METHOD 被定义为：

```
#define REGISTER_METHOD(methodName)          _addMethod(ms_methods, &CMethodFactory::Create(&objectType::methodName, objectType::GetClassTypeName(), #methodName))
```

根据函数的原型（参数以及返回值）我们定义了类似下图所示的若干 template 的 Create 函数，具体的我们支持不超过 8 个参数的函数。那么对于每一个 REGISTER_METHOD，根据具体的函数，相应的 Create 函数就会被 specialized，从而一个合适的 CMethodBase 的子类被创建。

```
template<typename R, class ParamType>
static CMethodBase& Create(R(*methodPtr)(ParamType), const char* className, const char* propertyName)
{
    typedef CGenericMethodStatic1<R, ParamType> MethodType;
    CMethodBase* pMethod = BEHAVIAC_NEW MethodType(methodPtr, className, propertyName);
    pMethod->AddResultHanlder<R>();
    return *pMethod;
}
```

```
template<typename R, class ParamType1, class ParamType2>
static CMethodBase& Create(R(*methodPtr)(ParamType1, ParamType2), const char* className, const char* propertyName)
{
    typedef CGenericMethodStatic2<R, ParamType1, ParamType2> MethodType;
    CMethodBase* pMethod = BEHAVIAC_NEW MethodType(methodPtr, className, propertyName);
    pMethod->AddResultHanlder<R>();
    return *pMethod;
}
```

当 Loading 某个 BT 的时候，

```
CMethodBase* LoadMethod(const char* value_)
```

LoadMethod 被调用，'value_' 是类似“Self.AgentParTest::FnFloatParam_In(float Self.AgentParTest::FloatProperty)”这样的字符串，根据 Method 的名字（class 名字及函数名字）从反射系统找到注册的相应的函数并且 clone 一份，需要 clone 的原因在于函数调用的具体场合的参数是不同的，这些参数需要被保存在 CMethodBase 里从而在执行的时候被使用。

下图是有两个参数并且有返回值这样的方法执行时的函数：

- 首先通过 GetValue 取得参数的当前值，如果参数是常数 const，它直接返回该值，如果参数是某个属性或 par，则取得其当前的值。
- 然后通过函数指针 m_methodPtr 调用该函数。
- 最后，函数执行结束后，函数的参数有可能被函数调用修改，通过 SetVariableRegistry 来修改相应的属性或 par。

```
virtual void run(const CTagObject* parent, const CTagObject* parHolder)
{
    const ParamBaseType1& v1 = this->m_param1.GetValue(parent, parHolder);
    const ParamBaseType2& v2 = this->m_param2.GetValue(parent, parHolder);

    R returnValue = (((ObjectType*)parent)->*this->m_methodPtr)(
        (PARAM_CALLEDTYPE(ParamType1))v1,
        (PARAM_CALLEDTYPE(ParamType2))v2);

    this->m_param1.SetVariableRegistry(parent, v1);
    this->m_param2.SetVariableRegistry(parent, v2);

    if (this->m_return)
    {
        *(behaviac::AsyncValue<R>*)this->m_return = returnValue;
    }
}
```

这样的执行过程似乎有些情况下有些冗余，不够最优，而事实确实如此。但是由于可以导出 Cpp 代码，而我们导出的 Cpp 代码则根据该节点函数调用的参数是否是 const，参数是否常量，参数是属性还是 par 等具体情况产生最优的代码。具体细节可以参考相应章节。Cpp 代码的生成

3 模板及类型的扩展

Behaviac 可以集成到不同的引擎，而不同的引擎有不同的数据类型，如何不修改 behaviac 而能够方便灵活的支持不同的数据类型呢？办法是 template！和类型相关的一共有一下若干类：

```
template<typename T> const char* GetClassTypeName(T*);
template<typename T> bool FromString(const char* str, T& val);
template<typename T> behaviac::string_t ToString(const T& val);
```

对于struct或class类型，DECLARE_BEHAVIAC_OBJECT_NOVIRTUAL宏自动添加了FromString和ToString的支持，对于enum，DECLARE_BEHAVIAC_OBJECT_ENUM宏也自动添加了支持。实际serialize的时候，如下图利用class traits这样的meta programming的技巧对struct或class或者enum类型做出分别的相应：

```
template<typename T>
inline bool FromString(const char* str, T& val)
{
    return Detail::FromStringStructHanler<T, behaviac::Meta::HasFromString<T>::Result>::FromString(str, val);
}
```

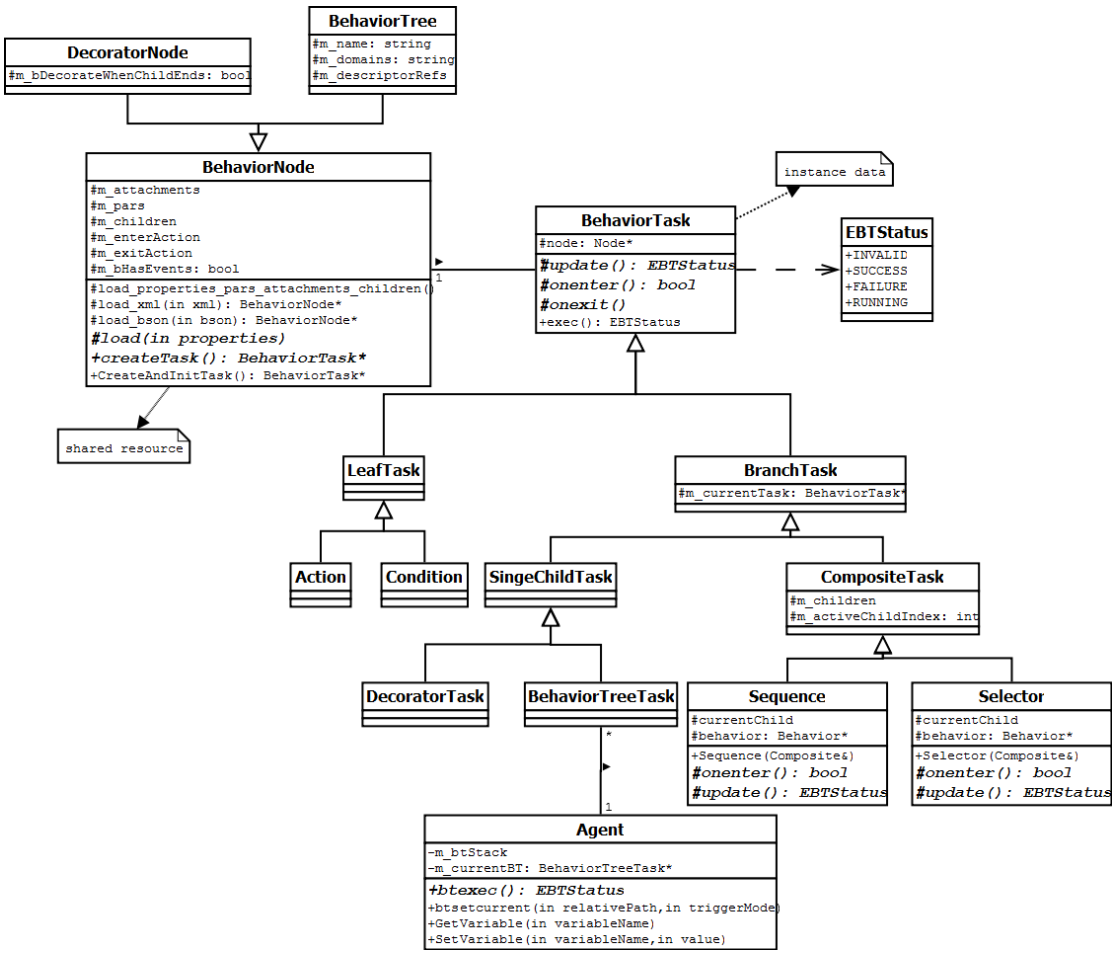
```
template<typename T>
inline behaviac::string_t ToString(const T& val)
{
    return Detail::ToStringStructHanler<T, behaviac::Meta::HasToString<T>::Result>::ToString(val);
}
```

关于扩展类型，可以参考教程‘Behaviac Tutorial’中的相关章节。

4 共享节点

在具体的游戏中，一个 Agent 类型可能有多份实例，出于对内存使用的考虑，behaviac 设计为资源和运行时两个部分来尽量减少对内存的占用。一个 BT 的拓扑结构，配置等都是属于类似资源的可以共享的部分，而运行时节点的状态等不能共享，不同的实例需要有不同的数据来保存。

4.1 类图



在上图中，BehaviorNode 及其子类是共享的资源类，仅需要一份。而 BehaviorTask 及其子类是根据 load 的 BT 树创建出来的每个 Agent 都需要有的，有很多份。

需要指出的是在该图中并没有完整的列出所有支持的子类，比如 DecoratorLoop，Conditon，Action 等等。

4.2 加载

```
/**
relativePath is relateve to the workspace exported path. relativePath should not include extension.
the file format(xml/bson) is specified by SetWorkspaceSettings.

@param bForce
force to load, otherwise it just uses the one in the cache
*/
static bool Load(const char* relativePath, bool bForce = false);
```

Workspace::Load(Agent::btload 实际上直接调用 Workspace::Load)负责加载（Load）BT 树。上图是 Workspace::Load 的原型。加载的 BT 树被添加到 Cache 中，后续通过 Workspace::Load 加载的时候，如果 ‘bForce’为 true，则强制读取文件重新加载，否则直接从该 Cache 中返回。

Workspace::Load 加载 BT 的时候，根据设定的格式选择使用 xml、bson 还是 cpp，c#等格式。

4.3 热加载

我们知道，在游戏运行的时候，`Workspace::SetWorkspaceSettings` 被调用来指定 workspace 的导出路径：

```
Workspace::SetWorkspaceSettings("../integration/unity/Assets/Resources/example_workspace/exported", format);
```

在开发版中，当热加载激活的时候（可以在 `Workspace::SetWorkspaceSettings` 中的第 3 个参数关闭）该导出路径被“监听”，当有文件被重新导出（xml、bson）的时候，并且如果指定的格式非 cpp 或 c#的时候，该 BT 被自动的重新加载，相应 Agent 实例中的 BT 树也被重新创建。如果指定的格式是 cpp 或 c#的时候不支持热加载。

4.4 更新包

当游戏已经发布后，可能会有更新包更新游戏，BT 可能被更新。

- 如果更新包可以包含可执行程序的话，更新的 BT 可以通过更新的 Cpp 文件或 C#文件从而 build 到更新的可执行程序里来更新。
- 但如果这个更新包只能是资源的时候，更新的 BT 就只能是更新的的 xml 或 bson 文件了。游戏使用更新包的逻辑是检查 BT 的导出路径，如果有更新的 xml 或 bson 文件，则使用之而不再使用原本 build 到 exe 的 BT 或原本的 BT。

5 执行

类 behaviac::Agent 几乎是 behaviac 中最重要的类，用户的 Agent 都需要从该类继承从而利用 behaviac 的功能。behaviac::Agent 有一个重要的函数 btexec 负责执行 BT：

```
/**
exec the BT specified by 'btName'. if 'btName' is null, exec the current behavior tree specified by 'btsetcurrent'.
*/
virtual EBTStatus btexec(const char* btName = 0, TriggerMode triggerMode = TM_Transfer);
```

btexec 执行当前的 BT，根据其执行的结果如果是 BT_SUCCESS 或 BT_FAILURE 则意味着当前的 BT 执行结束，如果在执行堆栈上有 BT，则弹出执行堆栈上最上面的那个 BT 作为当前的 BT。

5.1 子树及调用模式

一个 BT 可以调用其他 BT 或递归调用自己。或者是一个 BT 可以被某个事件中断从而执行其他 BT。这个时候原本运行的 BT 就被‘压’到执行堆栈上，而这个新的 BT 作为当前 BT。

当切换到这个新的 BT 的时候，有‘Transfer’和‘Return’两种模式：

- Transfer 模式，当前的 BT 被中断 abort 和重置 reset，新的 BT 被设置为当前 BT。
- Return 模式，当前的 BT 直接‘压’到执行堆栈上而不被 abort 和 reset，新的 BT 被设置为当前 BT，当这个新的 BT 结束的时候，原本的那个 BT 从执行堆栈上‘弹出’从当初的节点恢复执行。

作为子树被调用的时候，用的是 Return 模式。而被事件中断而执行中断子树的时候，缺省是 Transfer 模式，也可以选择 Return 模式。

5.2 执行一个 BT

通过函数 BehaviorTask::exec：

```cpp
EBTStatus BehaviorTask::exec(Agent* pAgent)
{
    bool bEnterResult = false;
    if (this->m_status == BT_RUNNING)
    {
        bEnterResult = true;
    }
    else
    {
        //reset it to invalid when it was success/failure
        this->m_status = BT_INVALID;

        bEnterResult = this->onenter_action(pAgent);
    }

    if (bEnterResult)
    {
        this->m_status = this->update(pAgent, BT_RUNNING);
        if (this->m_status != BT_RUNNING)
        {
            this->onexit_action(pAgent, this->m_status);
        }
    }
    else
    {
        this->m_status = BT_FAILURE;
    }

    EBTStatus currentStatus = this->m_status;

    return currentStatus;
}
```

执行一个节点。BehaviorTask 有下述 3 个虚函数：

```cpp
virtual EBTStatus update(Agent* pAgent, EBTStatus childStatus) = 0;
virtual bool onenter(Agent* pAgent);
virtual void onexit(Agent* pAgent, EBTStatus status);
```

函数 exec 首先执行 onenter，如果成功（返回 true），则继续执行 update，如果 update 返回非 BT_RUNNING，即 BT_SUCCESS 或 BT_FAILURE，则意味着该节点结束，则执行 onexit 并且返回该状态；否则返回 BT_RUNNING。当返回 BT_RUNNING，下一次执行 exec 的时候，则继续执行 update，如此重复一直到 update 返回非 BT_RUNNING，然后执行 onexit 结束该节点。

特别需要指出的是，虽然上述过程是对一个 BehaviorTask 来说的，但一个 BT 逻辑上也是一个 BehaviorTask，一个 BT 的执行逻辑也是一样的，只不过 update 的具体实现会有不同。

```cpp
EBTStatus DecoratorTask::update(Agent* pAgent, EBTStatus childStatus)
{
    EBTStatus status = super::update(pAgent, childStatus);

    if (!this->m_bDecorateWhenChildEnds || status != BT_RUNNING)
    {
        EBTStatus result = this->decorate(status);

        if (status != BT_RUNNING)
        {
            BehaviorTask* child = this->m_root;
            if (child)
            {
                child->m_status = BT_INVALID;
            }

            this->SetCurrentTask(0);
        }

        return result;
    }

    return BT_RUNNING;
}
```

```cpp
EBTStatus BranchTask::update(Agent* pAgent, EBTStatus childStatus)
{
    BEHAVIAC_UNUSED_VAR(childStatus);

    EBTStatus status = BT_INVALID;

    if (this->m_currentTask)
    {
        EBTStatus s = this->m_currentTask->GetStatus();
        if (s != BT_RUNNING)
        {
            this->SetCurrentTask(0);
        }
    }

    if (this->m_currentTask)
    {
        status = this->tickCurrentNode(pAgent);
    }

    return status;
}
```

如上图，分别是 DecoratorTask 和 BranchTask 的 update 的实现。对于一个 BranchTask，成员

`m_currentTask` 来表明执行状态为 BT_RUNNING 并且下一个 exec 时需要自己执行的子节点，如果有一个有效的 `m_currentTask`，则继续执行之直到该节点结束，则把执行结果返回该节点的父节点处理来决定接下来的执行情况。

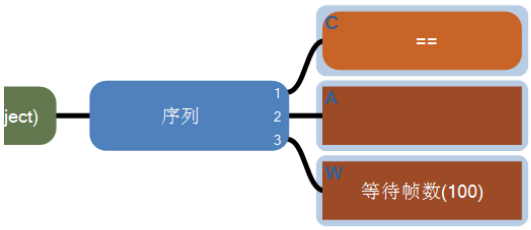
执行一个 BT 的 exec 总是返回 EBTStatus 的这样的一个状态，成功、失败或运行；返回成功可以认为是返回 true，返回失败可以认为是返回 false，有的时候认为返回 true 或 false 在上下文的理解上更方便些。

- 返回成功或失败意味着 BT 结束。下一次如果还被执行，则完全重新开始，即重新 enter，update，exit。
- 返回运行的情况则不同。如果本次执行返回运行，下次继续执行的时候，则继续执行（继续 update），如此循环直到返回成功或失败从而结束（exit）。特别的，该运行状态的节点在 BT 内部被‘记住’，下次执行的时候，‘直接’执行该运行状态的节点，而其他已经结束的节点不再被执行，理解这个处理对于把握 BT 的执行情况和效率比较关键。

由此可以看出 BT 就如同一个 coroutine，它 ‘知道’下一步从哪里继续。

5.3 更多的说明

假若某个 BT 如下图，一个序列有 3 个子节点，分别是一个条件节点 C，一个动作节点 A，一个等待节点 W。



条件节点对左右参数进行比较，

- 如果比较结果为 True，则条件节点返回成功 Success
- 如果比较结果为 False，则条件节点返回成功 Failure
- 条件节点不可能返回 Running

而动作节点则需要根据配置或实现来决定其返回值。
假若动作节点 A 执行后直接返回成功，而等待节点 W 是要等待 100 帧。C++代码调用 Agent::btexec 更新该 BT 的时候，如果代码类似下面的：

```
int frames = 1;
while (true) {
    BEHAVIAC_LOGINFO("Frame %d: behaviac::Agent::btexec", frames++);
    if (pAgent->btexec() != BT_RUNNING) {
        BEHAVIAC_LOGINFO("Finish");
        break;
    }
}
```

其执行过程则如下：

```
Frame 1: behaviac::Agent::btexec
    exec C
    exec A
    exec W

Frame 2: behaviac::Agent::btexec
    exec W

Frame 3: behaviac::Agent::btexec
    exec W

...

Frame 100: behaviac::Agent::btexec
    exec W

Finish
```

5.4 Log 文件

实际上，在 behaviac 中，当游戏运行时可以查看 exe 所在目录的_behaviac_$_.log，该文件类似如下图：

```
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->BehaviorTree[-1]:exit [success] [1]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->BehaviorTree[-1]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->BehaviorTree[-1]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Selector[1]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Selector[1]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Sequence[23]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Sequence[23]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Condition[30]:enter [success] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Condition[30]:update [running] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Condition[30]:exit [failure] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->Sequence[23]:exit [failure] [2]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->ReferencedBehavior[29]:enter [success] [1]
[4/23/2014 12:01:12 PM][tick]test_ns::AgentTest#test_ns.AgentTest_0_7 TestBehaviorGroup/scratch.xml->ReferencedBehavior[29]:update [running] [1]
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 bool par_alive->False
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 int par_enemy->0
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 int par_target->1
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 string par_category->
[4/23/2014 12:01:12 PM][property]test_ns::AgentTest#test_ns::AgentTest_0_7 int par_event_param->0
```

每一行的具体格式首先是用‘#’分割的类名和实例名。

需要说明的是，此外还有另一个 log 文件：_behaviac_$_$_.log，该文件是包含 behaviac 运行过程中的 info，warning 或 error 或其他信息的，不是包含 BT 执行情况的，BT 的执行情况统一输出到_behaviac_$_.log。

5.4.1 执行状态行

[tick]行是 BT 的执行情况，某个 Agent 实例上的某个 BT 上的某个节点的执行情况。

[tick]ClassFullName#InstanceName BTName->NodeClassName[NodeId]:ExecAction [EBTStatus] [Count]

在用‘#’分割的类名和实例名后，是 BT 树，接下来用‘->’作为分割，接下来是节点的类型名字及用’[]’括起来的节点 ID，接下来是用‘:’作为分割，之后是执行动作及用’[]’括起来的执行状态。最后是用’[]’括起来的的数字表明执行次数。

5.4.2 属性行

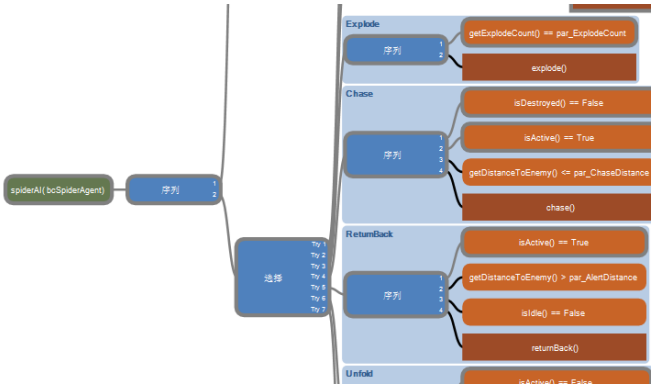
[property]行是某个 Agent 实例上的属性或 par 的值。

[property]ClassFullName#InstanceName Type Name->Value

在用‘#’分割的类名和实例名后，是数据类型，接下来用‘->’作为分割，接下来值。

5.5 一棵不好的行为树

下图所示 BT 某种意义上不是一颗“好”的 BT。该 BT 每次执行要么返回成功，要么返回失败。而一个“好”的 BT 在大部分情况下应该返回运行（Running），返回成功或失败意味着该 BT 结束了。所以很多情况下需要在 BT 里添加一些类似循环的节点。



这样做的原因在于运行（Running）状态的节点在下一次执行的时候会直接被执行，而一个返回成功或失败也就是结束了的 BT，下一次执行的时候还必须重新从头开始。

6 事件处理

任何一个 BT 都可以作为事件子树，作为 event 附加到任何的一个节点上(用鼠标拖动 BT 到节点)。当 Runtime 运行该 BT 的时候，如果发生了某个事件，可以通过 Agent::FireEvent 来触

发该事件，则处于 running 状态的节点，从下到上都有机会检查是否需要响应该事件，如果有该事件配置，则相应的事件子树就会被触发。
当某个事件发生的时候，通过 Agent::FireEvent 来触发 BT 的切换：

```
template<class ParamType1>
BEHAVIAC_FORCEINLINE void Agent::FireEvent(Agent* pAgent, const char* eventName, const ParamType1& param1)
{
    CNamedEvent* pEvent = 0;
    if (pAgent)
    {
        pEvent = pAgent->findEvent(eventName);

        if (!pEvent)
        {
            int contextId = pAgent->GetContextId();
            const CTagObjectDescriptor& meta = pAgent->GetDescriptor();
            pEvent = findNamedEventTemplate(meta.ms_methods, eventName, contextId);
        }

        if (pEvent)
        {
            CNamedEvent1<ParamType1>* pEvent1 = CNamedEvent1<ParamType1>::DynamicCast(pEvent);
            if (pEvent1)
            {
                pEvent1->SetParam(pAgent, param1);
            }
            else
            {
                BEHAVIAC_ASSERT(0, "unregistered parameters %s", eventName);
            }

            pEvent->SetFired(pAgent, true);
        }
        else
        {
            BEHAVIAC_ASSERT(0, "unregistered event %s", eventName);
        }
    }
}
```

上图所示代码是触发带有一个参数的事件，带有多个参数事件的触发和此类似。首先在 Agent 中找是否有该事件注册，如果找到，则把参数通过 SetParam 设置到该事件里，然后通过 SetFired 触发。

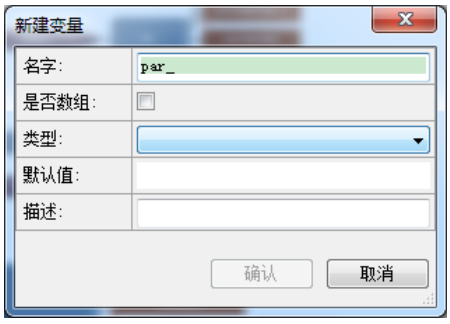
```
void SetParam(behaviac::Agent* pAgent, const ParamBaseType& param)
{
    //this->m_param = param;
    const char* eventName = FormatString("%s::%s::param0", this->m_classFullName, this->m_propertyName);
    pAgent->SetVariable(eventName, param);
}
```

SetParam 构造一个名字诸如“test_ns::AgentTest::Exploded::param0”，把参数赋给以该名字命名的一个“Variable”。而在 BT 中，如下图所示，实际上已经做了相应的“绑定”，从而当需要访问“par_event_param”的时候，程序能够“知道”访问命名为“test_ns::AgentTest::Exploded::param0”的“Variable”。

```
<pars>
  <par name="par_category" type="string" value="" />
  <par name="par_event_param" type="int" value="0" eventParam="test_ns::AgentTest::Exploded::param0" />
</pars>
```

7 par

par 是 behaviac 中一个重要的部分。par 是 parameter 的缩写，这里作为一个专有‘名词’来特别指代参数的使用。
par 非常类似 Blackboard，但 par 的优势是所有有关 par 的使用都是通过 GUI 进行的，用户必须首先创建某个指定类型的 par，然后在可以被使用的地方可以从下拉列表里根据需要的类型选择。
在编辑器中可以创建 par，每个 par 都有一个名字和类型，可以赋给初始值。



par 在编辑器中虽然是在具体的某个 BT 中创建的，而且也是存在 BT 中的：

```
<behavior name="dummy_test" agenttype="test_ns::AgentTest" version="1">
  <pars>
    <par name="a" type="int" value="0" />
    <par name="c" type="string" value="" />
  </pars>
```

但 par 在 Runtime 是通过 Agent 来管理的。当不同的 BT 出现了同名的 par 的时候，依然可以工作，而且是使之工作的一个有利之处。但如果该同名的 par 的类型不同，在运行时会有 assert 出现。如果 par 在 Runtime 中通过 Agent::SetVariable 设置了某个值，则该 par 在 BT 中就是该值，除非在 BT 中被设置了其他值。如下图中的使用：

```
pAgent->SetVariable("par_item", 0);

behaviac::Agent::SetIdMask(1);
pAgent->SetIdFlag(1);

LogManager::GetInstance()->SetEnabled(true);
LogManager::GetInstance()->SetFlush(true);

Workspace::SetWorkspaceSettings("../integration/unity/Assets/Resources/example_workspace/exported", format);

pAgent->btsetcurrent("circular");

const int kCount = 10;

for (int i = 0; i < kCount; ++i)
{
    pAgent->btexec();
    int p = pAgent->GetVariable<int>("par_item");
    BEHAVIAC_ASSERT(1 == p);
}
```

Agent 中有成员 `m_variables` 来管理这些 par 已经相关事务。

```
Variables                       m_variables;
```

需要指出的是，在 BT 里声明的 par 在执行该 BT 的时候，该 par 才会在该 Agent 里创建。如果是想给该 par 在代码里设初值，只需要直接调用 SetVariable 来给该 par 设值即可。在执行 BT 之前或者 SetVariable 之前调用 GetVariable 试图检查该 par 是否存在是‘非法’的，在 debug 版本下可能会有 assert。

8 代码的生成

behaviac 可以导出 xml 和 bson，然而也可以导出为 cpp 文件或 c#直接 build 到 exe 里，从而加载（load）的时间可以忽略，其执行效率也可以得到很大提高。

8.1 导出 cpp 文件

导出 cpp 文件主要涉及对 Agent 中属性和方法的访问，特别是当属性和方法非 public 的时候怎么可以不修改源码而访问呢？templae 函数又一次解决了这个问题！

```
#define ACCESS_PROPERTY_METHOD                  \
    template<typename T, typename R>            \
    R& _Get_Property_();                        \
                                                \
    template<typename T, typename R>                                \
    R _Execute_Method_();                                           \
    template<typename T, typename R, typename P0>                   \
    R _Execute_Method_(P0&);                                        \
    template<typename T, typename R, typename P0, typename P1>      \
    R _Execute_Method_(P0&, P1&);                                   \
    template<typename T, typename R, typename P0, typename P1, typename P2> \
    R _Execute_Method_(P0&, P1&, P2&);                              \
    template<typename T, typename R, typename P0, typename P1, typename P2, typename P3>                    \
    R _Execute_Method_(P0&, P1&, P2&, P3&);                                                                 \
    template<typename T, typename R, typename P0, typename P1, typename P2, typename P3, typename P4>        \
    R _Execute_Method_(P0&, P1&, P2&, P3&, P4&);                                                            \
    template<typename T, typename R, typename P0, typename P1, typename P2, typename P3, typename P4, typename P5>  \
    R _Execute_Method_(P0&, P1&, P2&, P3&, P4&, P5&);                                                       \
    template<typename T, typename R, typename P0, typename P1, typename P2, typename P3, typename P4, typename P5, typename P6> \
    R _Execute_Method_(P0&, P1&, P2&, P3&, P4&, P5&, P6&);                                                                     \
    template<typename T, typename R, typename P0, typename P1, typename P2, typename P3, typename P4, typename P5, typename P6, typename P7>  \
    R _Execute_Method_(P0&, P1&, P2&, P3&, P4&, P5&, P6&, P7&);                                                                              \
    template<typename T, typename R, typename P0, typename P1, typename P2, typename P3, typename P4, typename P5, typename P6, typename P7, typename P8>  \
    R _Execute_Method_(P0&, P1&, P2&, P3&, P4&, P5&, P6&, P7&, P8&);
```

每一个 Agent 及其子类都包含有上述宏。

对属性的访问通过 template 函数_Get_Property_<T， R>，不同的属性需要提供不同的类型 T，如下图，访问属性 HP 的时候，按照即定的规范生成 TPROPERTY_TYPE_framework_GameObject_HP，而代码生成器“知道”HP 的类型 R 是 unsigned int，从而可以正确的生成代码 _Get_Property_<framework::PROPERTY_TYPE_framework_GameObject_HP, unsigned int >。

```
struct PROPERTY_TYPE_framework_GameObject_age { };
template<> inline signed long& GameObject::_Get_Property_<PROPERTY_TYPE_framework_GameObject_age>()
{
    unsigned char* pc = (unsigned char*)this;
    pc += (int)offsetof(framework::GameObject, framework::GameObject::age);
    return *(reinterpret_cast<signed long*>(pc));
}

struct PROPERTY_TYPE_framework_GameObject_HP { };
template<> inline unsigned int& GameObject::_Get_Property_<PROPERTY_TYPE_framework_GameObject_HP>()
{
    unsigned char* pc = (unsigned char*)this;
    pc += (int)offsetof(framework::GameObject, framework::GameObject::HP);
    return *(reinterpret_cast<unsigned int*>(pc));
}
```

```
        unsigned int& opl = ((framework::GameObject*)pAgent_opl)->_Get_Property_<framework::PROPERTY_TYPE_framework_GameObject_HP, unsigned int >();
```

类似的对方法的访问则通过 template 函数_Execute_Method_<T,R,P0,...Pn>， n 是参数个数。不同的方法需要提供不同的类型 T，如下图，访问方法 distanceToPlayer 的时候，按照即定的规范生成 METHOD_TYPE_framework_GameObject_distanceToPlayer ， 而代码生成器 “知道” distanceToPlayer 的原型，即 R 是 float，没有参数，从而可以正确的生成代码 _Execute_Method_<framework::METHOD_TYPE_framework_GameObject_distanceToPlayer, float >()。

```
struct METHOD_TYPE_framework_GameObject_distanceToPlayer { };
template<> inline float GameObject::_Execute_Method_<METHOD_TYPE_framework_GameObject_distanceToPlayer>()
{
    return this->framework::GameObject::distanceToPlayer();
}

struct METHOD_TYPE_framework_GameObject_GoStraight { };
template<> inline void GameObject::_Execute_Method_<METHOD_TYPE_framework_GameObject_GoStraight>(signed int& p0)
{
    this->framework::GameObject::GoStraight(p0);
}

struct METHOD_TYPE_framework_GameObject_playerIsAligned { };
template<> inline bool GameObject::_Execute_Method_<METHOD_TYPE_framework_GameObject_playerIsAligned>()
{
    return this->framework::GameObject::playerIsAligned();
}

  float opl = ((framework::GameObject*)pAgent)->_Execute_Method_<framework::METHOD_TYPE_framework_GameObject_distanceToPlayer, float >();
```

8.2 导出 C#文件

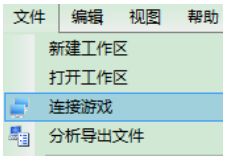
导出 C#文件和 Cpp 文件思路类似，不同之处在于 C#代码生成器“知道”属性是否是 public 或非 public，如果是 public，则直接访问，否则通过反射系统调用。

9 调试的实现

游戏运行中，打开编辑器可以和游戏建立连接，从而查看 BT 的执行情况以及属性或 par 的值，并且可以修改属性或 par 的值，还可以设断点等。

9.1 连接游戏

可以通过菜单



或快捷按钮连接游戏。

9.2 Runtime

要建立游戏和编辑器的连接，在 Runtime 需要在初始化或比较开始的地方调用 `behaviac::Socket::SetupConnection(bBlock)`，

- 如果‘bBlock’为 true，则游戏将被阻塞，编辑器和游戏建立连接后才能继续。
- 如果‘bBlock’为 false，则游戏不被阻塞，在游戏过程中，可以选择和游戏建立连接。
- 无论 bBlock’为 true 或 false，建立的连接在断开后都可以再次建立。
- 游戏和编辑器建立连接后，通过 TCP/IP 发送消息。所发的消息的格式和 log 文件中一致。可以参考 Log 文件。

```
/**
@param bBlocking
if true, block the execution and wait for the connection from the designer
if false, wait for the connection from the designer but doesn't block the game
*/
BEHAVIAC_API bool SetupConnection(bool bBlocking, unsigned short port = 60636);
BEHAVIAC_API void ShutdownConnection();
```

游戏和编辑器建立连接后，通过 TCP/IP 发送消息。所发的消息的格式和 log 文件中一致。可以参考 Log 文件。

所有的 BT 执行全部在 Runtime 端完成，编辑器之所以能够刷新来表现 BT 的执行情况，都是通过从游戏发送过来的消息而实现的。

10 Revision

2014 年 5 月 12 日

- 元信息
- 共享节点
- 执行
- 代码的生成
- 调试的实现

2014 年 5 月 6 日

- 初始版本